Semi-Annual Report

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

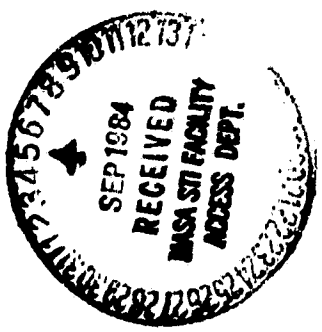
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Edmond H. Senn
ACD - Computer Science and Applications Branch

Submitted by:

John C. Knight
Professor

Report No. UVA/528213/CS85/105
July 1984



DEPARTMENT OF COMPUTER SCIENCE

The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements

Semi-Annual Progress Report

John C. Knight

Department of Applied Mathematics and Computer Science

University of Virginia

Charlottesville

Virginia, 22901

July 1984

CONTENTS

1. Introduction 1

2. Implementation Status 3

3. Sequencer Control Language 5

4. Ada And Hardware Fault Tolerance 13

5. Ada And Software Fault Tolerance 17

5.1 Exceptions 18

5.2 N-Version Programming 19

5.3 Recovery Blocks 21

5.4 Conversations 23

6. Professional Activities 27

References 29

Appendix 30

1. Introduction

The purpose of this grant is to investigate the use and implementation of Ada* in distributed environments in which reliability is the primary concern. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the software or underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

During this grant reporting period our primary activities have been:

(1) Continued development and testing of our fault-tolerant Ada testbed.

(2) Consideration of desirable language changes to allow Ada to provide useful semantics for failure.

---

* Ada is a trademark of the U.S. Department of Defense

(3) Analysis of the inadequacies of existing software fault tolerance strategies.

(4) The preparation of various papers and preparations.

Except for the sequencer, the various implementation activities of our fault-tolerant Ada testbed are described in section 2. The sequencer has been given a new, relatively sophisticated control language, and it is described in section 3.

In our analysis of the deficiencies of Ada, it has been quite natural to consider what changes could be made to Ada to allow it to have adequate semantics for handling failure. In section 4, we describe some thoughts on this matter reflecting what we consider to be the minimal changes that should be incorporated into Ada.

We consider it to be important that attention be paid to software fault tolerance as well as hardware fault tolerance. The reliability of a system depends on the correct operation of the software as well as the hardware. Software fault tolerance is rarely used in practice and when it is used, it is ad hoc with no formalism or organization. One of the reasons for this state of affairs is the general inadequacy of existing proposals for building software in a fault-tolerant manner. Before reviewing Ada and trying to incorporate software fault tolerance mechanisms into the language changes we consider necessary, we have reviewed the state of the art and prepared a systematic set of criticisms of existing proposal for the provision of fault tolerance in software. This set of criticisms is summarized in section 5.

During the grant reporting period we have made various reports about this work. Our activities in this area are described in section 6.

2. Implementation Status

We have continued our implementation activities of both the testbed and the associated translator. The translator translates a subset of Ada which includes most of the tasking and exception handling mechanisms into code for the virtual processors implemented by the testbed.

Some parts of the testbed have had to be redesigned and reimplemented as a result of obtaining a more accurate understanding of the way in which Ada operates. In many cases, the language definition is very obscure and it is quite difficult to determine exactly what is meant. In other cases, the semantics are comprehensible but extremely complex making an accurate implementation difficult. An area that has given us a great deal of difficulty is the exception mechanism. It appears relatively simple and straight forward as first but the many possibilities for exception generation during processing of declarations for example makes an accurate implementation very difficult. Our implementation of the exception mechanism has been redesigned and the implementation is being revised.

The overall state of the implementation can be gauged from the fact that the simple program that we have used as an example in various papers and presentations has been successfully executed using the translator and the testbed. The source text of the program that was executed is contained in the appendix of this report. A small number of other tests have been run and used to find errors in the translator and testbed. We are just beginning a systematic effort to debug the system.

The system continues to run on a single VAX using UNIX processes to simulate computers and UNIX pipes to simulate communications facilities. We had intended to use a network of IBM Personal Computers as the target of this testbed.

The use of the VAX/UNIX combination has always been viewed as an interim step that allowed us to develop the software in a relatively convenient and friendly environment. Clearly the facilities of the IBM PC are relatively limited although probably adequate with sufficient care. The major problem of porting the testbed to the IBM PC's would be the very long compile times resulting from the slow processor, the small memories, and the use of floppy disks.

Our department has been fortunate in receiving funds for the purchase of some Apollo workstations. We feel these computers might be more appropriate for the support of the testbed so we have chosen not to attempt porting the testbed until all the Apollo computers have arrived and been installed. They are equipped with relatively large main memories and hard disks, and in general are more powerful computers than the IBM PC's. The Apollo's also support a variant of UNIX which should allow porting the testbed quite simply. However, in order to avoid spending inordinate amounts of time investigating the idiosyncrasies of the Apollo system or building pieces of support software, we have decided to wait until other research projects have successfully used the Apollos and demonstrated that they can provide the facilities we need before attempting to use them.

3. Sequencer Control Language

Recall that the testbed is trying to allow experimenters to answer "what if..." questions about concurrent Ada programs. The sequencer control language is the experimenters interface with the testbed and so its form and facilities are extremely important.

Why is control of parallel programs any different from sequential programs? The reason is that "what if..." questions about tasking cannot be answered easily (sometimes never) because, in most implementations, a set of tasks cannot be forced into the necessary state that leads to the "what if..." question. This is not the case with sequential languages because they are deterministic. In most debugging systems for sequential languages there is a single-step facility whereby effects of individual instructions within a program can be studied in detail. Concurrent languages, on the other hand, are nondeterministic. There is no guarantee that a particular state of interest is reached on any given execution. For example, suppose a set of Ada tasks is executing asynchronously on the Ada testbed with the scheduler controlling which task runs when. The experimenter may be interested in asking questions such as: "What would happen if this particular task were forced into a certain state in its execution and this other task were forced to stop at a specific point in its execution?" and then "What do the contents of memory look like for a particular virtual processor at this point?". These questions are typical of those asked for controlling parallel programs. This is the level of control that is essential for the monitoring and experimentation of these Ada tasks. Hence, the main function of the command language is to provide the facilities for performing this control. Control is needed not only to single-step individual tasks, but to single-step them in relation to each other.

The command language interpreter provides the interface between the user command level and the sequencer module of the testbed. It receives the command line, interprets it, and passes the validated information to the rest of the sequencer which is then responsible for actually performing the actions to carry out these commands.

In the design of the sequencer command language, there are basically two elements essential to the design for control of Ada tasks. They are the ability to monitor, in some meaningful way, the tasking activity so as to understand the behavior of the parallel tasks, and the ability to perform experiments based, either implicitly or explicitly on the information gathered. Through the interaction of these two elements, the user can attempt to gain an understanding of the causes of existent errors or at least to note where the implementation and the expected behavior of the parallel tasks differ.

The overall strategy that is taken in the design of the command language is to control Ada tasks, not to debug Ada programs. First, the testbed must be viewed from an operational semantic definition standpoint: semantic in that it pertains to answering questions of language meaning; operational in that it allows programs to be executed and their actions to be observed. Furthermore, the definition must provide the ability to answer the "what if..." questions

Given these general requirements, we established the following minimal set of detailed requirements for control of the sequencer and hence the testbed:

(1) Starting a desired experiment. This requires the availability of the compiled Ada code to be interpreted and the map showing how the abstract processors for the experiment are to be mapped to physical processors.

(2) Executing named tasks. This requires a list of the task names (any number) that the experimenter wishes to start executing. This command was originally separate but it has been included with the command for restarting tasks which have been stopped. This was done since the involved tasks are each at their own fixed code location and the one command for starting could then be viewed as a set of tasks being suspended at a particular breakpoint (breakpointing being the ability to temporarily halt an executing program); for the initial starting up of a task's execution then this breakpoint would be defined at location zero. The start would always be from a current breakpoint.

(3) Exiting from the existing test environment. A provision must be made to allow the experimenter to have a summary of important system information listed upon exit.

(4) Stopping or artificially suspending named tasks no matter what they are doing. As with starting task execution, a list of the tasks, again any number, the user wishes to stop or suspend must be given. A common example of a situation that would use this command would be one in which there was the desire to observe temporary suspension of all but one process in order to eliminate interference from any of the other processes.

(5) Causing a particular abstract processor (AP) to fail. Since a major point of the testbed is to see if software strategies can tolerate processor failures, the experimenter should be provided with the ability to fail any processor. Giving the AP number of the particular AP to be failed would cause the physical processor owning the subject AP to cease to schedule it.

(6) Setting and unsetting breakpoints. The general problem regarding breakpoints involves the desire to have tasks suspended in the middle of statements. Since AP code may be shared among tasks, specification of breakpoints by location only is insufficient. Therefore, a breakpoint has to be defined such that it is named by the source-level task name (task id) and a code location. It is also considered desirable that the effects of a breakpoint be delayed so that a task must execute that code location more than once before "hitting" the breakpoint. This latter facility is required to provide more flexibility to the user and his desire to perform experiments with loops or end conditions

(7) Restarting tasks' executions. As described above, a list of task names would be given to start or resume any number of tasks executing. This would allow the named tasks to run until they encounter a breakpoint or terminate. The ability to restart task execution is important because many fault-tolerant strategies call for automatic replacement of defective hardware.

(8) Single stepping a particular task. This would require the name of the task that is to be involved and the number of instructions that are to be executed before the subject task is temporarily halted; absence of the count should yield a default of single stepping the named task through the interpretation of exactly one instruction. This capability would allow a user to deal with tasks through a perspective which is more microscopic than the Ada source language level. For instance, each process can be brought to the desired state by executing to a breakpoint set for that process and single stepping for fine adjustment from there.

(9) Displaying the sequencer's tables. These displays would provide a quick and useful reference of which tasks are running, where and what there current

breakpoints are, etc.

(10) Displaying the state of the testbed's data structures. This level of control would be valuable in decisions that must be made regarding branches. The user could breakpoint before the branch, display the memory contents and decide what to do next on the basis of that. All of these display capabilities would provide the means of monitoring whether the fault-tolerant strategy that is being tested works or not.

(11) Calling upon a help facility. This would permit the user at any time before, during, or after the experiment to view the available commands that are allowed; syntax and usage of each command would be provided.

(12) Recalling commands. This would allow the experimenter to look at a log of commands that he has used.

With this set of command facilities, the experimenter will have a good basis for implementing the kind of control that is needed in a first, elementary, but useful control mechanism for Ada tasks. It satisfies the two elements initially described as essential to the control of Ada tasks: it posseses commands to allow the ability to monitor the tasking activity at a microscopic level and it provides the ability at any moment of the inspection to perform experiments as to the future endeavors of those tasks. This set is by no means complete and there exists a lot of remaining issues that require investigation before further expansion of the control mechanism can be made.

Listed below are the actual commands of the command language interpreter as presently implemented:

NEW

Start an experiment. The names of the files containing the AP to PP map and program must be given.

QUIT

Exits an experiment without having a summary dump listed.

QUITD

Exits an experiment and has a summary dump listed.

RESUME

Starts or resumes any number of tasks executing; execution will stop when a breakpoint is hit. The names of the tasks to be resumed must be listed; a "*" in place of the task name list will resume all currently started tasks.

STOP

Stops any number of tasks executing. The names of the tasks to be stopped must be listed; an "*" in place of the task name list will stop all tasks that are running.

KILL

Causes one AP to be killed (failed). The AP number to be killed must be given.

BREAK

Sets a breakpoint according to a named location. The task name and address in the task at which to set the breakpoint must be given. An optional count may be given to indicate the number of times to execute the instruction before stopping occurs; the default is one.

UNSET

Unsets a breakpoint according to a named location. The task name must be

given. An optional code offset may be given to indicate the address in the task at which the breakpoint was set. If no code offset is given, all breakpoints for that task are unset.

SINGLESTEP

Executes the named task one instruction at a time for the given number of instructions. The task name must be given. A count is optional to give the count of instructions to execute with the default being one.

DISPLAY APTOPPMAP

Displays the AP_number to PP_number map.

DISPLAY VPTOAPMAP

Displays the VP_name to AP_number map.

DISPLAY TASKTOVPMAP

Displays the task_id to VP_name table.

DISPLAY VPDATASTRUCTURE

Displays the VP data structure.

DISPLAY VPSTATE

Displays the state and location of the named VP.

DISPLAY BREAKPOINTS

Displays all of the breakpoints in the current experiment.

HELP

Displays all of the available commands with the ability to give a description of each.

FLASHBACK

Displays the last specified number of commands. If no number is provided it

defaults to 15.

The command language interpreter also provides in its command language several other capabilities and features including abbreviations for the commands, good error handling and feedback of the error messages to the user, checks made on all parameters, a UNIX-like MORE facility for certain commands like the flashback command, and sensible screen layouts.

4. Ada And Hardware Fault Tolerance

We have summarized our concerns about Ada's inability to deal with processor failure by pointing out that the problem is basically one of omitted semantics. Nothing is stated in the Ada Language Reference Manual about how programs are to proceed when a processor is lost in a distributed system although the manual does specifically include distributed computers as valid targets.

We have proposed additional semantics to deal with this situation. The heart of these additional semantics is the notion that the loss of a processor and consequently the loss of part of the program can be viewed as equivalent to the execution of abort statements on the lost tasks. Thus in all cases, failure semantics would be equivalent to the semantics of abort.

We have also proposed a comprehensive mechanism for implementing these semantics. This mechanism requires quite extensive changes to the execution-time support for Ada but it is feasible as we have shown in our testbed implementation.

The use of abort semantics is not the most elegant approach. There are numerous consequences that seem rather extreme if considered out of context. For example, abort semantics imply that all the dependent tasks of a task that is lost must be terminated even if they are still executing on non-failed computers. The *overwhelming* advantage of abort semantics is that they do not require that the language be changed.

A more elegant and clearly preferable approach in the long run is to modify the language and to introduce language structures that include appropriate failure semantics. During the grant reporting period we have been considering what form these language structures might take.

Although Ada ignores this problem, other languages do not and language designers have proposed various schemes in the literature. For example, Liskov has proposed "guardians" [1], and "atomic actions" [2] have been proposed by several people. We have considered both along with other schemes, as candidates for inclusion in Ada. None of these proposals seem appropriate however because they are not able to provide the performance level that is required in the kind of applications for which Ada is intended. The naive introduction of atomic actions into Ada would reduce performance substantially; probably making the language worthless.

Given that language structures with more sophisticated semantics probably cannot be added to Ada, we have considered what more modest changes could be made that would be in the spirit of the language but would provide acceptable performance. We have broken the lack of failure semantics in Ada into two parts and addressed each separately. The two parts are *entrapment in communication* and *loss of context*, both of which we have documented extensively in the past.

Entrapment in communication can be dealt with in a revised language much like it is with abort semantics. Raising an exception in a task that is the subject of entrapment is a reasonable way to inform the task of the problem and to provide a mechanism to allow it to proceed. The difficulty that follows from something like this is the subsequent difficult with redirection of communication. Given that a task has been lost and cannot be used in further communication, it is necessary to communicate with its alternate. Since Ada (as presently defined) requires that the caller explicitly use the name of the callee in a rendezvous, a different call must be used for the alternate. This means that all communication must be guarded (probably by an IF statement) so that different entry calls can be made. This is a

large burden to put on the programmer, and it can hardly be described as elegant. We have no well-defined suggestions on preferable language structures at this time.

We also observe that the Ada rendezvous makes no provision for broadcast messages. There are plenty of occasions when a single task needs to communicate with a whole set of other tasks; for example starting a set of real-time services or informing a set of tasks about machine failure at the level of the application software. This seems like a serious omission.

The loss of context problem is actually far more serious. With abort semantics, loss of context requires that parts of the program be removed when this may not be strictly necessary. One solution that we have considered is to require that the general nesting structure of the program be reflected in the way tasks are assigned to processors. For example, only tasks at the outermost level would be the subject of controlled distribution. All nested tasks would be required to be assigned to the same processor as their parent. This seems like a reasonable solution since any loss of context takes with it all the objects that could reasonably use that context. It is however a major restriction on the forms that programs may take.

The key problem with this type of limitation is that it may not be suitable at all for certain applications. Consider for example a system which includes a special-purpose hardware processor; a fast-fourier transform unit for example. The Ada code which provides access to the services of this unit will obviously reside on the unit. The fast-fourier transform functions may be required from many parts of the program but the programmer might be reluctant to make these routines global. Good programming practice may well dictate that such routines be nested. Allowing nested objects to be distributed seems almost mandatory.

In considering this problem we have concluded that it really is essential to be able to locate nested objects separately from their parents. To solve the resulting loss of context problem, we propose that Ada's scope rules be enhanced to include objects that are distributable and have limited scope. We propose that the objects to be distributed be a new form of package and that the scope of objects in the package be limited to that package only. Access to the package would be through the objects made visible in the specification of the package in the usual way.

Our consideration of this topic is not complete. We will continue to look at desirable extensions to Ada and complete the definition of the enhanced communications mechanism and the distributable packages.

5. Ada And Software Fault Tolerance

We have examined the literature on fault-tolerant software with the goal of determining the adequacy of Ada in providing a software fault tolerance mechanism. We find that Ada makes *no provision whatsoever* for software fault tolerance. Consequently we plan to consider what extensions to Ada might be desirable to support fault-tolerant software.

In examining the literature we have concluded that the schemes that have been proposed are inadequate in general and in many cases incomplete. In this section we review the inadequacies of previous work in software fault tolerance.

A general consideration for crucial systems is time. Boolean acceptance tests and voting codes must be reached and reached on time for the results to be useful at all. A common problem, which we refer to as *the unexpected delay problem*, is that some unanticipated circumstance, e.g. an infinite loop, may cause a particular section of code to be executed too late for its results to be useful or not to be executed at all. If a scheme does not address the unexpected delay problem, then it is insufficient for providing software fault tolerance in a real-time program since a program in that context needs only to be late to be considered faulty. Another consideration for a fault-tolerance scheme is the management of complexity. If the use of a scheme involves too much effort on the designer's (programmer's) part, it may be counter-productive in that more faults will be generated through the use of the scheme than would otherwise occur. Furthermore, a fault in the application of a fault-tolerance scheme might make the system more dangerous than if fault tolerance efforts had not been applied at all. A scheme supported by a rigid, encasing, structured syntax allows design-time (compile-time) enforcement of the accompanying semantic rules. Such a quality in a scheme allows for added

complexity without added faults.

5.1. Exceptions

Although claimed to be suitable for software fault tolerance, exception handling can only deal effectively with anticipated faults, not the unanticipated faults addressed by an actual fault-tolerance approach. A crucial system should have anticipated faults removed before it is placed into service. Exceptions can be used within systems to represent and deal with expected, normal, but unusual situations.

In most languages, but particularly in Ada, when an exception handler is entered there is no indication of exactly from where control transferred. Neither is there an indication of how much of the state has been damaged. These problems make it difficult for a handler either to repair the fault and transfer back to the point where the exception was raised, or to replace the execution of the remainder of the "procedure".

Often the finite list of available exception names (even when user defined names are included) is very general, such as in Ada: range_check, numeric_error, constraint_error, and tasking_error. As a result, the exception could have been raised in any of many statements (components), or in one of many places in one statement. Consider, for example, the following statement:

```
I := A(J) + B(K) + C(L) + D(M);
```

If the execution of this statement raises a subscript error, there are four different subscript that could be involved. Also note that the subscript violation is a

symptom of the actual fault. The actual fault might lie in the calculation of J or K or L or M, or it might be in some decision computation that erroneously directed control to this statement. Further, attempts to determine the extent of the damage by examining values in the state could raise another exception. Since one fault existed in the routine covered by the handler, it cannot be assumed that no others will exist in a continuation that attempts the same algorithm. Since multiple faults may have existed in that part of the routine already executed, ascribing the erroneous state detected to one fault and "handling" that one may not correct the state at all. Indeed, if the fault to which the detected error is ascribed is not one of the actual faults in the routine, the actions of the handler may cause even more damage.

Exception handling involves predicting or enumerating the faults that may occur in a system so a handler can be provided for each. This may be impracticable in a complex system. A failure to predict an exception and provide a handler for it could bring about the collapse of the entire control system or at the very least wreak havoc within some part of it. If a handler is provided for an exception with the expectation that that exception was only to be raised in one portion of a routine, but it was actually raised in another portion or propagated up from a component routine, the actions of the handler could be entirely inappropriate.

5.2. N-Version Programming

Although the method employs parallelism, it still implements software fault tolerance in logically sequential parts of a system: It is not a concept or construct for dealing with parallel programs.

The n-version programming proposals all assume that all versions will arrive at the cross-check points -- they ignore the unexpected delay or infinite loop problem.

The proponents of n-version programming claim that the scheme is inherently more reliable than, say, recovery blocks. The reliability of the scheme depends upon the reliability of the voting criteria and test for agreement. That is just as volatile as the recovery block's acceptance test. How to actually do the voting is unspecified. There are discussions of different choices for dealing with single numerical values, such as weighted sums, but not for the general case of a vector of values of differing types. The discussions on voting on single numerical results concludes that that is very difficult, but most applications are going to need long vectors of results of differing types. It would seem that voting in an actual control system might become impossible. The proponents have admitted that n-version programming may not be applicable in many situations [3].

The n-version programming strategy depends upon the ability to create independent versions or programs derived from the same specification. As for how the independence of versions is to be achieved, there are appeals to the use of independent programming teams using different languages. Problems may arise from common programming experience and current fashions in algorithms, or even from a specification that specifies too much.

As for the use of different programming languages and translators, that can be a source for faults. Translators for different programming languages are likely to use incompatible representations for even the simplest data structures, and will certainly provide incompatible synchronization mechanisms. The software that attempts to rectify these differences in preparation for distribution of inputs and

gathering and voting upon results, either becomes a bottleneck subject to single-point failure or must itself be made fault-tolerant. If that software is made fault-tolerant by n-version programming, the software providing the same service for it comes into question, ad infinitum.

Implementing an n-version program is not as easy as the descriptions make it out to be. It appears at first easy to do n-version programming in Ada — just put each version in its own task and let them execute. But problems arise in obtaining the results in order to vote on them and even in ensuring that all or most versions even reach the cross-check points! Infinite loop problems can occur, and arranging for a faulty task to consent to a rendezvous with the driver is no mean feat. Voting in general presents a centralized bottleneck and is therefore undesirable for distributed applications.

5.3. Recovery Blocks

Since the recovery block concept relies on syntactic support from the programming language in use, and Ada fails to provide this syntax, recovery blocks cannot be used in Ada as presently defined. However, there are fundamental technical problems with recovery blocks also and we review them in this section.

In a recovery block, there is only one test for acceptability of results. How to program the acceptance test to be both meaningful and allow a wide range of alternate algorithms to pass it is unspecified. Design diversity in the primary and the alternates, combined with the possibility of degraded service from the alternates, implies that the acceptance test must not be made very strict. It must be possible for any results of the primary or any alternate (assuming they are

correct) to pass the test, yet it must be strict enough to detect errors produced by any of the primary or the alternates. This combination may not be possible. A test that is general enough to pass all valid results might not be specific enough to actually detect all errors within the construct. The strategies involved in the primary and in the many alternates may be so divergent as to require separate checks on the operation of each "try" as well as an overall check for acceptability as regards the goal of the statement. The recovery block really needs multiple tests, one for the primary and one specific to each of the alternate algorithms, perhaps with a general overall test as a check on the various individual tests.

Like n-version programming, the recovery block scheme depends upon the generation of independent versions of software, in this case, to be used as the primary and alternates. Due to the degraded service concept, the alternates do not have to produce results so close as to be able to vote upon them, but they also need a certain degree of independence to reduce the possibility that they will contain the same or very similar faults. How to get independent versions for alternates is not really addressed in the recovery block proposals.

The recovery block is strictly a sequential programming construct. It gives no hint about recovery after inter-process communication. The conversation concept is an appropriation of the recovery block concept, not an integral part.

There is the question of when a recovery block should be used. There is little indication as to what portions of a program should be protected by recovery blocks. If used on every routine and every statement sequence, the tests may become trivial and fail to offer any benefit. If recovery blocks are only used at the outermost levels, the acceptance tests may be so complex as to duplicate the complexity of the primary or alternates. This may introduce more faults in the

acceptance test than the primary alone, or it may squander processing resources so that execution of an alternate would bring about a timing failure.

The infinite loop problem and its generalization, the timing of control program activities has remained unaddressed by the recovery block scheme.

How can we rectify the use of unrecoverable objects with the backward recovery strategy? There is some discussion in the literature on how recovery blocks could be reconciled with nested recovery block commitment to unrecoverable objects.

The problem of the latency intervals for fault detection being longer than commitment intervals is not addressed. That is related to the problem of how to construct meaningful acceptance tests. It is assumed that acceptance tests can be constructed that can detect errors before they become so wide spread, or that multiple layers of nested recovery blocks' acceptance tests can together detect them. The possibility of nested recovery blocks allowing such errors to "escape" should not be permitted.

5.4. Conversations

As with recovery blocks, the use of conversations requires programming language support. Again, Ada fails to provide any but this is not too surprising since there are no satisfactory proposals in the literature. This is one of the major shortcomings of conversations.

Conversations have been criticized in the past for failing to provide a mechanism preventing "desertion". Desertion is the failure of a process to enter a conversation when other processes expect its presence. Whether the process will

never enter the conversation, is simply late, or enters the conversation only to take too long or never arrive at the acceptance test(s), does not matter to the others if they have deadlines to meet, as is likely in a crucial system. Thus, desertion is another form of what we have called the infinite loop problem. The processes in a conversation must be extricated if the conversation begins to take too long. Each process may have its own view of how long it is willing to wait, especially since processes may enter a conversation asynchronously. Also, a deserter can be considered erroneous, but determining which process is a deserter could be difficult. Only the concurrent recovery block scheme even addresses the desertion problem. The solution there is to enclose the entirety of each participating process within the conversation. Not only can a process fail to arrive at a conversation, it cannot exist outside of the conversation.

The original conversation proposal made no mention of what was to be done if the processes ran out of alternates. Two presumptions may be made: that the retries proceed indefinitely, which is inappropriate for a real-time system, or that an error is to be automatically detected in each of the processes, as is assumed in all of the proposed conversation syntaxes. What the syntactic proposals do not address is that, when a process fails in a primary attempt at communication with one group of processes to achieve its goal, it may want to attempt to communicate with an entirely different group as an alternate strategy for achieving that goal. This is the kind of divergent strategy alluded to above. The name-linked recovery block and the conversation monitor schemes do not mention whether it is an error for different processes to make different numbers of attempts at communicating. Although they may assume that is covered under the desertion issue, that may not necessarily be true if processes are allowed to converse with alternate groups.

Russell's work [4] permitting the application to have direct control over establishment, restoration, and discard of recovery points has its own set of problems. First of all, his premise ignores the possibility that the information within a message can contaminate a process' state. When the receiver of a message is rolled back, he merely replaces the same message on the message queue. This was the main "advantage" derived from knowing the direction of message transmission. His application area is that of producer-consumer systems. The control systems we are considering are feedback systems. A producer almost always wants to be informed about the effects of the product, and a consumer almost always wants to have some influence over what it will be consuming in the future. The relationships between sensors and a control system and between a control system and actuators can be viewed as pure producer-consumer relationships but sensors and actuators are more accurately modeled as unrecoverable objects. The scheme allows completely unstructured application of the MARK, RESTORE, and PURGE primitives. This fact, along with the complicated semantics of conversations, which they are provided to create, affords the designer much more opportunity to introduce faults into the software system.

All of Kim's proposals [5] use monitors for inter-process communication. In a distributed system, monitors and any other form of shared variables are vulnerable to extensive delays. While a monitor may be implemented as a fully-replicated distributed database, most other implementations leave its information vulnerable to processor failure. With an independently executing process, as one would simulate a monitor in Ada, the application could decide upon appropriate times to save copies for use by a replacement after reconfiguration. But the traditional monitor is not active and long periods may pass without any process calling a

procedure that updates a replacement monitor's state.

Since the name-linked recovery block proposal makes no mention of the method of communication among processes within a conversation, it remains open to charges of permitting smuggling. If processes use monitors, message buffers, of ordinary shared variables, other processes can easily "reach in" to examine or change values while a conversation is in progress. Kim also states that ensuring proper nesting of name-linked recovery blocks is impossible.

The conversation monitor is designed to prevent smuggling but, as Kim's description stands, it allows a problem that is even more insidious than smuggling. A monitor used within a conversation is initialized for each use of the conversation, but not for each attempt within a conversation. This allows partial results from the primary or a previous alternate to survive state restoration within the individual processes. Since such information is in all probability erroneous, it is likely to contaminate the states within this and all subsequent alternates.

Our conclusion from all of this is that Ada makes no provision for fault-tolerant software but that none of the proposed technologies are really complete and ready for use. Extensive work is needed to complete the theory before practical use can be made in Ada and similar programming languages.

6. Professional Activities

During the grant reporting period we have prepared several papers and made various presentations about this work.

We were invited to a workshop sponsored by Westinghouse Space and Electronics Center in Baltimore Maryland. The purpose of the workshop was to allow Westinghouse personnel to become familiar with various technologies for crucial systems, and to expose researchers to the present and pending DoD-related projects requiring very high reliability.

We were also invited to participate in a panel session at the Distributed Processing conference held in San Francisco in May. This panel addressed distributed Ada and the other panel members were David Fisher from Gensoft Corporation, Robert Firth from Tartan Laboratories, Bryce Barton from Hughes Aircraft, and Dennis Cornhill from Honeywell. There was some agreement among the panelists and substantial disagreement. Nothing that was said affected our position on the inadequacies of Ada for distributed computing.

In the 1983annual report for this grant we included copies of two papers that had been submitted to the Fourteenth Fault-Tolerant Computing Systems Symposium (FTCS 14). One of those papers (appendix 3 in the report) was rejected. We disagree with many of the comments made by one of the referees and have written to the conference organizers requesting clarification. The second paper (appendix 4 in the report) was accepted and was presented at FTCS 14.

We have revised the paper rejected by FTCS 14 and submitted it to the IEEE Computer Society's Fourth Symposium on Reliability in Distributed Software and Database Systems. We have also prepared a paper entitled Difficulties With Ada As A Language For Reliable Distributed Processing and submitted it to the IEEE

Computer Society's 1984 Conference on Ada Applications and Environments. Both of these papers have been supplied to the sponsor separately from this report.

A lengthy paper describing most of our work on Ada in some detail was being prepared when we submitted our 1983 annual report. A preliminary version of that paper was included in that report as appendix 5. That paper has been completed and submitted to the IEEE Transactions on Software Engineering. We are awaiting an editorial decision from that journal.

REFERENCES

(1) B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3.

(2) S. K. Shrivastava, "Structuring Distributed Systems for Recoverability and Crash Resistance," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 4.

(3) L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Digest of Papers FTCS-6*,

(4) D. L. Russell and M. J. Tiedman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9*, June 1979.

(5) K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based On Monitors," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3.

APPENDIX

```
procedure EXAMPLE is

  task CALLER is
    pragma distribute_to(1);
    pragma priority(1);
  end CALLER;


  task SERVER is
    entry E;
    pragma distribute_to(2);
    pragma priority(1);
  end SERVER;

  task ALTERNATE_SERVER is
    entry ABNORMAL_START;
    entry E;
    pragma distribute_to(1);
    pragma priority(1);
  end ALTERNATE_SERVER;

  task body CALLER is
    SYSTEM_STATE  : integer;
  begin
    SYSTEM_STATE  := 1;
    write(1,1);
    loop
      MAIN_BLOCK:
      begin
        if SYSTEM_STATE  = 1 then
          write(1,2);
          SERVER.E;
          write(1,3);
        else
          write(1,4);
          ALTERNATE_SERVER.E;
          write(1,5);
        end if;
      exception
        when TASKING_ERROR=>
          SYSTEM_STATE  := 2;  -- abnormal
      end MAIN_BLOCK;
    end loop;
  end CALLER;
```

```
task body ALTERNATE_SERVER  is
begin
  write(2,1);
  accept ABNORMAL_START;
  loop
    write(2,2);
    accept E;
    write(2,3);
  end loop;
end ALTERNATE_SERVER;


task RECONFIGURE_1 is
  entry FAILURE(WHICH  : in integer);
  pragma distribute_to(1);
  pragma priority(2);
  for FAILURE use at 10;
end RECONFIGURE_1;

task body RECONFIGURE_1 is
begin
  loop
    write(3,1);
    accept FAILURE(WHICH  : in integer) do
      write(3,2);
      ALTERNATE_SERVER.ABNORMAL_START;
      write(3,3);
    end FAILURE;
  end loop;
end RECONFIGURE_1;


task ALTERNATE_CALLER  is
  entry ABNORMAL_START;
  pragma distribute_to(2);
  pragma priority(1);
end ALTERNATE_CALLER;

task body ALTERNATE_CALLER  is
begin
  write(4,1);
  accept ABNORMAL_START;
  loop
    write(4,2);
    SERVER.E;
    write(4,3);
  end loop;
end ALTERNATE_CALLER;
```

```
task body SERVER is
begin
  write(5,1);
  loop
    write(5,2);
    accept E;
    write(5,3);
  end loop;
end SERVER;


task RECONFIGURE_2 is
  entry FAILURE(WHICH  : in integer);
  pragma distribute_to(2);
  pragma priority(2);
  for FAILURE use at 10;
end RECONFIGURE_2;

task body RECONFIGURE_2 is
begin
  write(6,1);
  accept FAILURE(WHICH  : in integer) do
    write(6,2);
    ALTERNATE_CALLER.ABNORMAL_START;
    write(6,3);
  end FAILURE;
end RECONFIGURE_2;


begin
  null;
end;
```